Fault-Tolerant Consensus in Unknown and Anonymous Networks

Carole Delporte-Gallet, Hugues Fauconnier and Andreas Tielmann
LIAFA, University Paris VII, France
{cd,hf,tielmann}@liafa.jussieu.fr

Abstract

This paper investigates under which conditions information can be reliably shared and consensus can be solved in unknown and anonymous message-passing networks that suffer from crash-failures. We provide algorithms to emulate registers and solve consensus under different synchrony assumptions. For this, we introduce a novel pseudo leader-election approach which allows a leader-based consensus implementation without breaking symmetry.

1. Introduction

Most of the algorithms for distributed systems consider that the number of processes in the system is known and every process has a distinct ID. However, in some networks such as in wireless sensors networks, this is not necessarily true. Additionally, such networks are typically not totally synchronous and processes may suffer from failures such as crashes.

Designing protocols for such networks is especially intricate, since a process can never know if its messages have been received by all processes in the system. In this paper, we investigate under which conditions information can be reliably shared and consensus can be solved in such environments.

Typically, in systems where no hardware registers are available, one makes additional assumptions to be able to reliably share information, e.g. by assuming a correct majority of processes. However, these techniques assume also some knowledge about the total number of processes. With processes with distinct identities, the requirements to emulate a register have been precisely determined by showing that the quorum failure detector Σ is the weakest failure detector to simulate registers in asynchronous message passing systems [5]. But again, this approach fails due to the lack of identities in our anonymous environment.

To circumvent these problems, we assume that the system is not totally asynchronous, but assume the existence of some partial synchrony. We specify our environments by using the general round-based algorithm framework (GIRAF) of [11]. This has two advantages: (i) it is easy to precisely specify an environment and (ii) it makes it easy to emulate environments to show minimality results.

We first define the moving source environment (MS) in which at every time at least one process (called the source) sends timely messages to all other processes, but this source may change over time and infinitely often. Although this environment is considerably weaker than a total synchronous environment, we show that it is still sufficient to implement registers, although it is not possible to implement the consensus abstraction. In fact, it can be emulated by hardware registers in totally asynchronous "known" networks for any number of process crashes. Therefore, if we would be able to implement consensus in this environment, we could contradict the famous FLP impossibility result [7]. This result states, that consensus cannot be implemented in asynchronous message passing networks, even if only one process may crash. Since we can emulate registers if only one process may crash [2], we can also emulate the MS environment and therefore cannot be more powerful.

To implement consensus, we consider some additional stronger synchrony assumptions. Our first consensus algorithm assumes that additionally to the assumptions of the MS environment, eventually all processes communicate timely. We call this environment the eventual synchronous (ES) environment. It resembles Dwork et al. [6]. In our second consensus algorithm, we consider a weaker environment and only assume that eventually always the same process is able to send timely to all other processes. We call it the eventual stable source environment (ESS). It resembles the model of [1] in which it is used to elect a leader, a classical approach to implement in turn consensus.

Due to the indistinguishability of several processes that behave identical, a true leader election is not possible in our anonymous environment. Therefore, in our second algorithm, we take benefit of the fact that it suffices for the implementation of consensus if all processes that consider itself as a leader behave the same way. We show how to eventually guarantee this using the history of the processes proposal values.

Furthermore, we consider the weak-set data-structure [4]. This data-structure comes along some problems that arise with registers in unknown and anonymous networks. Every process can add values to a weak-set and read the values written before. Contrary to a register, it allows for sharing information without knowing identities of other processes

Carole Delporte-Gallet and Hugues Fauconnier were supported by grant ANR-08-VERSO-SHAMAN. Andreas Tielmann was supported by grants from Région Ile-de-France.

and without the risk of an overwritten value due to a concurrent write. Furthermore, we show that it precisely captures the power of the MS environment, i.e. we can show that it can be implemented in the MS environment and a weak-set can be used to emulate the MS ennvironment. Interestingly, in known networks, a weak-set is equivalent to the register abstraction and can thus be seen as a generalization for unknown and anonymous networks.

Furthermore, we show that although it is possible to emulate registers in our MS environment, it is not possible to emulate Σ [5], the weakest failure detector for registers. And this result is not only due to the anonymity of the processes, it holds even if the number of processes and their identities are known. Note that this is not a contradiction, since the result in [5] means only that Σ is the weakest of all failure detectors with which a register can be implemented and we have exhibited synchrony assumptions where the existence of a failure detector is not necessary at all.

1.1. Related work

There have been several approaches to solve fault-tolerant consensus in anonymous networks deterministically. In [4], fault-tolerant consensus is solved under the assumption that failure detector Ω [3] exists, i.e. exactly one correct process eventually knows forever that it is the leader. In [9], fault-tolerant and obstruction-free[1] consensus is solved if registers are available.

There has also been some research on systems where IDs are known but the number of processes is not. In [8], it is assumed that processes may crash, but furthermore that it is possible to detect the participants initially. In [12], a leader election algorithm for a system where infinitely many processes may join the system is presented if the number of processes simultaneously up is bounded.

To the best of our knowledge, this paper presents completely new approaches to emulate registers and solve the consensus problem in unknown and anonymous environments with partial synchrony.

2. Model and Definitions

We assume a network with an unknown (but finite) number of processes where the processes have no IDs (i.e. they are totally anonymous) and communicate using a broadcast primitive. The set of processes is denoted Π. We assume that the broadcast primitive is reliable, although it may not always deliver messages on time. Furthermore, any number of processes may crash and the processes do not recover. Processes that do not crash are called correct.

1. For obstruction-free consensus, termination is only guaranteed if a process can take enough steps without beeing interrupted by other processes.

We model an algorithm A as a set of deterministic automata, one for every process in the system. We assume only fair runs, i.e. every correct process executes infinitely many steps.

2.1. Consensus

In the consensus problem, the processes try to decide on one of some proposed values. Three properties have to be satisfied:

Validity:	Every decided value has to be a proposed value.
Termination:	Eventually, every correct process decides.
Agreement:	No two processes decide different values.

2.2. An extension to GIRAF

Algorithm 1 presents an extension to the generic round-based algorithm framework of [11] (GIRAF). It is extended to deal with the particularities of our model, namely the anonymity and unknown number of the processes. The framework is modeled as an I/O automaton. To implement a specific algorithm, the framework is instantiated with two functions: *initialize*() and *compute*(). The *compute*() function takes the round number and the messages received so far as parameters. We omit to specify a failure detector output as parameter (as in [11]), because we are not interested in failure detectors here. Both functions are non-blocking, i.e. they are not allowed to wait for any other event.

Our extension lies in the way we model the received messages. Since the processes have no IDs, we represent the messages that are received during one round as a set instead of an array.

The communication between the processes proceeds in rounds and the advancement of the rounds is controlled by the environment via the *receive*$_i$ and *end-of-round*$_i$ input actions. These actions may occur separately at each process p_i and therefore rounds are not necessarily synchronized among processes. The framework can capture any asynchronous message passing algorithm (see [11]).

Environments are specified using round-based properties, restricting the message arrivals in each round.

2.3. Environments

We say that a process p_i is in round k, if there have been k invocations of *end-of-round*$_i$. A process p_i has a *timely link in round* k, if *end-of-round*$_i$ occurs in round k and every correct process p_j receives the round k message of p_i in round k.

In this paper, we consider three different environments:

- In the first one, which we call the moving-source (MS) environment, we assume that in every round k, there

Algorithm 1: Extended GIRAF generic algorithm for process p_i.

```
1  States:
2      k_i ∈ ℕ, initially 0;
3      M_i[ℕ] ⊆ Messages, initially ∀k ∈ ℕ : M_i[k] = ∅;
4  Actions and Transitions:
5      input end-of-round_i
6          if (k_i = 0) then
7              m := initialize();
8          else
9              m := compute(k_i, M_i);
10         M_i[k_i + 1] := M_i[k_i + 1] ∪ {m};
11         k_i := k_i + 1;
12     output send(⟨M_i[k_i], k_i⟩)_i;
13     input receive(⟨M, k⟩)_i
14         M_i[k] := M_i[k] ∪ M;
```

exists a process p_s (a source) that has a timely link in round k.

- In the second environment, which we call the eventual synchronous (ES) environment, we demand the same as in the MS environment, but additionally require that there is some round k such that in every round $k' \geq k$, all correct processes have timely links in round k'.
- In the third environment, which we call the eventually stable source (ESS) environment, we demand the same as in the MS environment, but additionally require that eventually the source process p_s is always the same in every round. This means, that there is some round k such that in every round $k' \geq k$, the same process p_s has a timely link in round k'.

3. Implementing consensus in ES

Algorithm 2: A consensus algorithm in ES for process p_i.

```
1  on initialization do
2      VAL := initial value;
3      WRITTEN := WRITTENOLD := PROPOSED := ∅;
4      return PROPOSED;
5  on compute(k_i, M_i) do
6      WRITTEN := ⋂_{m∈M_i[k_i]} m;
7      PROPOSED := (⋃_{m∈M_i[k_i]} m) ∪ PROPOSED;
8      if (k_i mod 2 = 0) then
9          if (PROPOSED = WRITTENOLD = {VAL}) then
10             decide VAL; halt
11         else if (WRITTEN ≠ ∅) then
12             VAL := max(WRITTEN);
13         PROPOSED := {VAL};
14     WRITTENOLD := WRITTEN;
15     return PROPOSED;
```

Algorithm 2 implements consensus in the ES environment. The idea of the algorithm is to ensure safety by waiting until a value is contained in every message received in a round. In this way, one can ensure that a value has also been relayed by the current source and is therefore known by everybody (we say that the value is written). If a process evaluates Line 9 to true, then VAL is known by everybody (because it was written in the last round) and no other process will consider another value as written, because only a value which has also been relayed by a source can be in WRITTEN. But the relayed value of a source would also be in PROPOSED at every process.

To guarantee the liveness of the consensus algorithm, we use the fact that eventually, all proposal values in the system are received in every even round by everybody and everybody will select the same maximum in Line 12. Therefore, everybody will propose the same value in the next round and the algorithm will terminate.

3.1. Analysis

For all local variables VAR, we denote by VAR_i the local variable of process p_i (e.g., PROPOSED_i). For every variable VAR_i, VAR_i^k is the value of this variable after process p_i has executed Line 7 when *compute* has been invoked with parameter k (i.e. in round k).

Lemma 1. *If no process has decided yet and for some p_i, $v \in \text{WRITTEN}_i^k$, then every process p_j that enters round k has $v \in \text{PROPOSED}_j^k$.*

Proof: If a process p_i has a value v in WRITTEN_i^k, then v has been contained in every message, which p_i has received in round k (Line 6). This includes the message of the source, since by assumption the source has not yet terminated. But by definition, every other process p_j that enters round k also has received the message of this source in this round and added it to its set PROPOSED_j^k (Line 7). Therefore, v is in PROPOSED_j^k. □

Lemma 2. *If no process has decided yet and p_i has $v \in \text{WRITTENOLD}_i^k$ in an even round k, then every other process p_j that enters round k has $v \in \text{WRITTEN}_j^k$.*

Proof: If a process p_i has a value v in WRITTENOLD_i^k, then it has had v in WRITTEN_i^{k-1}. Therefore, every other process p_j that enters round $k-1$ has v in PROPOSED_j^{k-1} in the same odd round $k-1$ (Lemma 1). Since no value is removed from a set PROPOSED in odd rounds, v will be contained in every set PROPOSED broadcast at the end of round $k-1$ and therefore get into WRITTEN_j^k at every process p_j that enters round k. □

Theorem 1. *Algorithm 2 implements consensus in the ES environment.*

Proof: We have to prove the 3 properties of consensus. Validity is immediately clear, because VAL is always an initial value.

To prove termination, assume that the system has stabilized, i.e. all faulty processes have crashed and all messages are received in the round after which they have been sent. Then, all processes receive the same set of messages in every round. Therefore, the set PROPOSED *and thus* WRITTEN *is the same at all correct processes and everybody will always select the same maximum in Line 12. In the next round all processes start with the same proposal value and this value will be written in every future round. Thus, everybody will evaluate Line 9 to true in the next round.*

To prove agreement, assume p_i is the first process that decides a value v in a round k. This means, that p_i has evaluated Line 9 to true. If some other value than v would have been written anywhere in the system, this would contradict PROPOSED $= \{v\}$ *(Lemma 1), since p_i is the first process that decides. Furthermore, v is in* WRITTEN *at every process in the system in round k, since it is also in* WRITTENOLD *(Lemma 2). Therefore, every other process decides v in the same round, or it will evaluate Line 11 to true and select v as new* VAL*. Thus, no other value will ever get into* PROPOSED *anywhere in the system, no other value will ever be written and no other value will ever be selected as* VAL*.*

□

4. Implementing consensus in ESS

Algorithm 3 implements consensus in the ESS environment. For the safety part, the algorithm is very close to algorithm 2 (see Section 3).

To guarantee liveness, we use the fact that we have at least one process which is eventually a source forever. We use the idea of the construction of the leader failure detector Ω [3]. It elects a leader among the processes which is eventually stable. In "known" networks, with some eventual synchrony, Ω can be implemented by counting heartbeats of processes (e.g. in [1]). But we are not able to count heartbeats of different processes here, because in our model the processes have no IDs. To circumvent this problem, we identify processes with the history of their proposal values. If several processes have the same history, they either propose the same value, or their histories diverge and will never become identical again. Eventually, all processes will select the same history as maximal history and the processes with this history will propose in every round the same values.

4.1. Implementation

Every process maintains a list of the values it broadcasts in every round (specifically, its proposal values). This list is denoted by the variable HISTORY. In this way, two processes that propose in the same round different values will eventually have different HISTORY variables. Note that, although the space required by the variables may be unbounded, in every round they require only finite space. Thus, if we could ensure that eventually all processes that propose have in every round the same history (and at least one process proposes infinitely often), then the proposal values sent are indistinguishable from the proposal values of a single "classical" leader.

However, the history of a process permanently grows. Therefore, every process includes its current history in every message it broadcasts. Furthermore, it maintains a counter C for every history it has yet heard of (in such a way that no memory is allocated for histories it has not yet heard of). Then, it compares the histories it receives with the ones it has received in previous rounds. If some old history is a prefix of a new history, it assigns the counter of the new history the value of the counter of the old one, increased by one. Thus, the counter of a history that corresponds to an eventual source is eventually increased in every round.

In this way, it is possible to ensure that eventually only eventual sources that converge to the same infinite history consider itself as leader. In a classical approach, eventually only these leaders would propose values. But to meet our safety requirements, it is crucial to ensure that all processes propose in every round at least something to make sure that the value of the current source is received by everybody. Therefore, we let processes that do not consider itself as a leader propose the special value $\perp$.

4.2. Analysis

Similarly to Section 3, for every variable VAR_i, VAR_i^k is the value of this variable after process p_i has executed Line 9 in round k.

Definition 1. *We say, that p_i* has heard of *p_j's round k message (m_j^k), if p_i has received m_j^k in round k, or if there exists another process p_l such that p_i has heard of p_l's round k' message for some $k' > k$ and p_l has heard of p_j's round k message.*

Let process p_s be an eventual source. We then identify three groups of processes:

- *out-connected*: The processes, the eventual source p_s has infinitely often heard of.
- *$\diamond$-silent*: The processes that are not *out-connected.*
- *$\diamond$-proposer*: The *out-connected* processes that have eventually in every round timely links towards all other *out-connected* processes.[2]
- *leader*: We say that a process p_i is a *leader* in some round k ($p_i \in$ *leader*(k)), iff $\forall \text{H}, \text{C}_i^k[\text{HISTORY}_i^k] \geq \text{C}_i^k[\text{H}]$.
  If process p_i is eventually a leader forever, i.e. there exists a k, such that for

2. Note that it is possible that the message an *out-connected* process actually has received is not the message that a *$\diamond$-proposer* has sent. It is sufficient if it receives an identical message from another process.

Algorithm 3: The consensus algorithm in ESS for process p_i.

```
1  on initialization do
2      VAL := initial value; ∀H, C[H] := 0; HISTORY := VAL;
3      WRITTEN := WRITTENOLD := PROPOSED := ∅;
4      return m = ⟨PROPOSED, HISTORY, C⟩;
5  on compute(k_i, M_i) do
6      WRITTEN := ⋂_{m∈M_i[k_i]} m.PROPOSED;
7      PROPOSED := ( ⋃_{m∈M_i[k_i]} m.PROPOSED) ∪ PROPOSED;
8      ∀H, C[H] := min_{m∈M_i[k_i]}(m.C[H]);
9      ∀m ∈ M_i[k_i], C[m.HISTORY] := 1 +
       max{ C[H] | H is a prefix of m.HISTORY};
10     if (k_i mod 2 = 0) then
11         if (WRITTENOLD = {VAL}) ∧ (PROPOSED ⊆
           {VAL, ⊥}) then
12             decide VAL; halt;
13         else if (WRITTEN \ {⊥} ≠ ∅) then
14             VAL := max(WRITTEN \ {⊥});
15         if
           (∀H, C[HISTORY] ≥ C[H])∨(PROPOSED ⊆ {VAL, ⊥})
           then
16             PROPOSED := {VAL};
17         else
18             PROPOSED := {⊥};
19     WRITTENOLD := WRITTEN;
20     WRITTEN := PROPOSED;
21     append VAL to HISTORY;
22     return m = ⟨PROPOSED, HISTORY, C⟩;
```

all $k' \geq k$, $p_i \in$ *leader*(k'), then we simply write that $p_i \in leader$. Note that it may be possible that there are several processes in *leader*.

The sets relate to each other in the following way:

$$\{p_s\} \subseteq \diamond\textit{-proposer} \subseteq \textit{out-connected} \subseteq \textit{correct}$$
$$\text{and} \quad \diamond\textit{-silent} \cap \textit{out-connected} = \emptyset$$

We will later show that *leader* $\subseteq$ $\diamond$*-proposer* (Lemma 6).

Lemma 3. *Eventually, in every odd round k, for every $\diamond$-proposer p_i, the set* PROPOSED *in m_i^k is a subset of the set* WRITTEN *at all out-connected processes in round $k+1$. More formally:*

$$\exists k, \forall k' \geq k \textit{ with } k' \bmod 2 = 1,$$
$$\forall p_i \in \diamond\textit{-proposer}, \forall p_j \in \textit{out-connected}:$$
$$m_i^{k'} = \langle \text{PROPOSED}, -, - \rangle$$
$$\rightarrow \text{PROPOSED} \subseteq \text{WRITTEN}_j^{k'+1}$$

Proof: Follows directly from the definition of $\diamond$-proposers and the fact that out-connected processes eventually do not receive any timely messages from $\diamond$-silent processes. □

Lemma 4. *Eventually, at all out-connected processes, the counters that correspond to histories of $\diamond$-proposers increase in every round by one. More formally:*

$$\exists k, \forall k' \geq k, \forall p_i \in \diamond\textit{-proposer}, \forall p_j \in \textit{out-connected},$$
$$\mathrm{C}_j^{k'+1}[\text{HISTORY}_i^{k'+1}] = \mathrm{C}_j^{k'}[\text{HISTORY}_i^{k'}] + 1$$

Proof: Assume a time when the system has stabilized. This means, that all $\diamond$-proposers send timely messages to all out-connected processes in every round and no out-connected process receives timely messages from $\diamond$-silent processes. Then, let k be the number of the current round and for every $\diamond$-proposer p_i let p_j be an out-connected process, such that the counter $\mathrm{C}_j^k[\text{HISTORY}_i^k]$ is minimal among all out-connected processes in round k. Then, the counter for p_i's history at p_j will never decrease, because p_j will never receive a message with a lower counter from any other process.

Since p_i is a $\diamond$-proposer, the counter for p_i's history will increase by one at p_j in every round. For every other out-connected process, since it receives also a message from p_i in every round and it can only finitely often receive a lower counter corresponding to p_i's history (the lowest one is p_j's), the counter of p_i's history eventually increases in every round by one. □

Lemma 5. *If a history of a process p_j infinitely often corresponds to a maximal counter at a $\diamond$-proposer p_i, then p_j is a leader forever. More formally:*

$$\forall p_i \in \diamond\textit{-proposer}, \forall p_j \in \Pi:$$
$$(\forall k, \exists k' > k, \forall h, (\mathrm{C}_i^{k'}[\text{HISTORY}_j^{k'}] \geq \mathrm{C}_i^{k'}[h]))$$
$$\rightarrow \; p_j \in \textit{leader}$$

Proof: We first show that $p_j \in \diamond$-proposer. Assume that it is not. Since $p_i \in \diamond$-proposer, eventually the counter that corresponds to p_i's history is increased by one at every out-connected process (Lemma 4). Since $p_j \notin \diamond$-proposer, some out-connected process p_l does not receive m_j^k in round k for infinitely many rounds k. Therefore, the counter at p_l that corresponds to p_j's history is not increased by one in these rounds and is eventually strictly lower than the one that corresponds to p_i's history. Since every time some out-connected process has a lower counter than the others, eventually this counter propagates to all other out-connected processes, p_i's history will eventually be higher than p_j's at all out-connected processes. A contradiction.

If p_i and p_j are both $\diamond$-proposers, then eventually they receive their messages timely in every round k. Since p_j's history increases at all out-connected processes by one (Lemma 4), eventually $\mathrm{C}_j^k[\text{HISTORY}_j^k] = \mathrm{C}_i^k[\text{HISTORY}_j^k]$. Since by our assumption, in some future round k', p_j's history is maximal at p_i and a counter can increase by at most one and the counters that correspond to p_j's

history increase always by one (Lemma 4), $C_j^k[\text{HISTORY}_j^k]$ *is maximal forever and therefore* p_j *is a leader forever.* □

Lemma 6. *Eventually, there exists a process* $p_i \in$ *leader and every leader is a* $\diamond$*-proposer. More formally:*

$$\exists k, \exists p_i \in \Pi, \forall k' \geq k : p_i \in \textit{leader}(k') \quad (1)$$

$$\textit{and} \ \ \forall p_i \in \Pi : (\forall k, \exists k', k' > k, p_i \in \textit{leader}(k')) \rightarrow \ p_i \in \diamond\textit{-proposer} \quad (2)$$

Proof: The eventual source p_s *is a* $\diamond$*-proposer. Therefore, there exists at least one* $\diamond$*-proposer. Either* p_s *is also a leader forever, or there is another process whose history infinitely often corresponds to a higher counter at* p_s *than* p_s*'s history. Then, with Lemma 5 this process is a leader forever. This implies (1).*

Assume a process p_i *is not a* $\diamond$*-proposer. Then,* p_i*'s counter is increased by less than one in infinitely many rounds at some processes. Because eventually these counters propagate to all out-connected processes and the values of* $\diamond$*-proposers are increased in every round by at least one (Lemma 4), eventually the history of some* $\diamond$*-proposer is higher than that of* p_i*. Therefore,* p_i *cannot be a leader forever. This implies (2).*

□

Lemma 7. *If no process has decided yet, then eventually only values of leaders and* $\perp$ *get into a set* WRITTEN *anywhere. More formally:*

$$\exists k, \forall k' \geq k, \forall p_i \in \Pi : \text{WRITTEN}_i^{k'} \subseteq \cup_{p_j \in \textit{leader}(k')} \text{VAL}_j^{k'} \cup \{\perp\}$$

Proof: There is a time after which there exists at least one leader and all leaders are $\diamond$*-proposers (Lemma 6) and since leaders propose their values always, all their values get into every set* WRITTEN *at all out-connected processes in every even round (Lemma 3).*

Therefore, every set PROPOSED *contains a value of a leader (compare Lemma 1) and no process that considers itself not as leader and has a value different from a leader will evaluate line 15 to true and add a different value to its set* PROPOSED. □

Theorem 2. *Algorithm 3 implements consensus in ESS.*

Proof: We have to prove the 3 properties of consensus. Validity is clear, since VAL *is always an initial value.*

To prove termination, assume there exists a run where no process ever decides. Then, eventually only non-$\perp$ *values of leaders will get into a set* WRITTEN *anywhere (Lemma 7) and they will get into* WRITTEN *always in every even round (Lemma 3) and all out-connected processes select the same value (the maximum in Line 14). Therefore, only this value and* $\perp$ *will be written in subsequent rounds and every out-connected process will select this value as value for* PROPOSED *in Line 16 (i.e., no out-connected process will select* $\perp$*) and everybody will evaluate Line 11 to true in the next round. Therefore, eventually, every correct process will decide.*

To prove agreement, assume p_i *is the first process that decides a value* v *in a round* k*. This means, that* p_i *has evaluated Line 11 to true. Then, as* PROPOSED $\subseteq \{v, \perp\}$*, no other value different from* $\perp$ *is in a set* WRITTEN *anywhere in the system (compare Lemma 1) and* v *is in* WRITTEN *at every process in the system in round* k*, since it is also in* WRITTENOLD *(compare Lemma 2). Therefore, every other process decides* v *in the same round, or it will evaluate Line 13 to true and select* v *as new* VAL *and no other value different from* $\perp$ *will ever get into* PROPOSED *anywhere in the system and therefore, no other value will ever be selected as* VAL.

□

5. Weak-Sets

The weak-set data structure has been introduced by Delporte-Gallet and Fauconnier in [4].

A weak-set S is a shared data structure that contains a set of values. It is defined by two operations: the $add_S(v)$ operation to add a value v to the set and the get_S operation which returns a subset of the values contained in the weak-set. Note that we do not consider operations to remove values from the set. Every get_S operation returns all values v where the corresponding $add_S(v)$ operation has completed before the beginning of the get_S operation. Furthermore, no value v' where no $add_S(v')$ has started before the termination of the get_S operation is returned. For add_S operations concurrent with the get_S operation, it may or may not return the values. Therefore, weak-sets are not necessarily linearizable[3].

5.1. Weak-Sets and registers

A weak-set is clearly stronger than a (regular) register:

Proposition 1. *A weak-set implements a (regular) multiple-writer multiple-reader register.*

Proof: To write a value, every process reads the weak-set and stores the content in a variable HISTORY*. Then, every process adds the value to be written together with* HISTORY *to the weak-set.*

To read a value, a process reads the weak-set and returns the highest value among all values accompanied by a HISTORY *with maximal length.*

This transformation satisfies the two properties of regular registers, namely termination and validity. Termination follows directly from the termination property of weak-sets.

If several processes write at the same time, two reads at two different processes may return different values, but after

3. A weak object is linearizable (also called atomic) if all of its operations appear to take effect instantaneously [10].

all writes have completed, the return value will be the same at all processes. To see that also validity holds, consider the value returned by a read. If there is no concurrent write, then the value returned is the last value written (i.e. the maximal value of all values concurrently written).

□

In [4], a weak-set is implemented using (atomic) registers in the following two cases:

Proposition 2. *If the set of processes using the weak set is known (i.e. the IDs and the quantity), then weak-sets can be implemented with single-writer multiple-reader registers.*

Proposition 3. *If the set of possible values for the weak set is finite, then weak-sets can be implemented with multiple-writer multiple-reader registers.*

5.2. Weak-Sets and the MS environment

Algorithm 4 shows how to implement a weak-set in the MS environment. Similarly to Section 3, for every variable VAR$_i$, VAR$_i^k$ is the value of this variable after process p_i has executed Line 15 in round k (i.e. after *compute* is called with parameter k).

Algorithm 4: A weak-set algorithm in the MS environment for process p_i.

```
1  on initialization do
2      VAL := ⊥; PROPOSED := WRITTEN := ∅;
3      BLOCK := false;
4      return PROPOSED;
5  on get do
6      return PROPOSED;
7  on add(v) do
8      PROPOSED := PROPOSED ∪ {v};
9      VAL := v;
10     BLOCK := true;
11     wait until (BLOCK = false);
12     return ack;
13 on compute(k_i, M_i) do
14     WRITTEN := ⋂_{m∈M_i[k_i]} m;
15     PROPOSED := (⋃_{m∈M_i[k'],1≤k'≤k_i} m) ∪ PROPOSED;
16     if (VAL ∈ WRITTEN) then BLOCK := false;
17     return PROPOSED;
```

Lemma 8. *If for some p_i, $v \in$ WRITTEN$_i^k$, then every process p_j that enters round k has $v \in$ PROPOSED$_j^k$.*

Proof: The proof is analogous to Lemma 1. □

Lemma 9. *If some value is in* WRITTEN *at some process, then this value will be forever in* PROPOSED *at all processes.*

Proof: Since it is never a value removed from any set PROPOSED*, this follows immediately from Lemma 8.* □

Theorem 3. *Algorithm 4 implements a weak-set.*

Proof: We have to show that all operations terminate at all correct processes and that every get operation returns all values which have been added before.

The only position where an operation may be blocked is in Line 11. But since eventually all messages will be received by all correct processes, every value will eventually be in every set PROPOSED *and therefore eventually be in every set* WRITTEN*. Thus, no correct process will block in Line 11 forever.*

To show that every get operation returns all values which have been added before, see that an add(v) operation only terminates if v is in WRITTEN *at some process. Together with Lemma 9, this means that this value will be returned by every process in Line 6.*

□

5.3. Emulation of the MS environment with weak-sets

Algorithm 5 emulates the MS environment using a weak-set S and the corresponding *add*$_S$ and *get*$_S$ methods.

As a weak-set is implementable by only using registers (see Proposition 2) and the FLP impossibility result [7] states that consensus is not implementable using only registers, this implies, that it is not possible to implement consensus in the MS environment (without any additional assumptions like in ES).

Algorithm 5: Emulating the MS environment for process p_i using a weak-set S.

```
1  on initialization do
2      DELIVERED := ∅;
3      trigger end-of-round_i;
4  on send(m_i, k_i)_i do
5      add_S(⟨m_i, k_i⟩);
6      forall ⟨m, k⟩ ∈ get_S \ DELIVERED do
7          DELIVERED := DELIVERED ∪ {⟨m, k⟩};
8          trigger receive(m, k)_i;
9      trigger end-of-round_i;
```

Theorem 4. *Algorithm 5 emulates the MS environment.*

Proof: Clearly, eventually all messages get delivered and all correct processes execute an infinite number of rounds.

It remains to show, that in every round k, there exists a process s_k such that for every process p_i at which end-of-round$_i$ occurs in round k, p_i receives the round k message of s_k in round k.

Let p_i be the first process that finishes to add the value of a round k. If several processes finish to add their values at exactly the same time, choose one.

***Claim:** Every process at which end-of-round is triggered in round k has received p_i's round k value.*

The proof is by contradiction. Assume that a process p_j triggers end-of-round in round k without having received p_i's round k value. By the definition of a weak-set, this means that p_j's get$_S$ begun before p_i's add$_S$ was completed. But a process will only start a get$_S$ after it has finished to add its own value. A contradiction to the fact that p_i was the first process that has completed its add$_S$. □

6. The MS-environment and the Σ failure detector

The quorum failure detector Σ [5] outputs lists of IDs of trusted processes (i.e. it is not well-defined in our anonymous model) and it satisfies the following properties:

Intersection:	Given any two lists of trusted processes, possibly at different times and by different processes, at least one process belongs to both lists.
Completeness:	Eventually at all correct processes, every trusted process is correct.

Σ has been shown to be the weakest failure detector to emulate registers in totally asynchronous message-passing systems [5] (with known IDs). This means, that Σ is sufficient to emulate registers in such systems and with any failure detector which is also sufficient to implement registers in such a system, it is possible to emulate Σ. Interestingly, although it is possible to implement a register in the MS environment (via weak-sets), we show that even if we assume that the number of processes and their IDs are known, it is not possible to emulate Σ. Note that this is no contradiction, since in our model no failure detector is necessary for the emulation.

Proposition 4. *It is not possible to emulate Σ in the MS-environment, even if the number of processes and their IDs are known.*

Proof: Assume there exists such an algorithm and consider a run r_1 where process p_1 is the only correct process, p_1 is always the source, and p_1 receives no messages from other processes. Then, by the completeness property of Σ, there exists some time t after which the output of Σ is $\{p_1\}$.

Similarly, consider a run r_2 where process p_2 is the only correct process and p_1 crashes after time t. Again, p_1 is the source until time t and receives no messages from other processes (this is possible, since the messages from p_2 may be arbitrary delayed). For p_1, run r_1 and r_2 are indistinguishable up to time t and consequently the Σ at p_1 will output $\{p_1\}$ at p_1 at time t. But since eventually, the output at p_2 has to be $\{p_2\}$ forever, this contradicts to the intersection property of Σ.

□

7. Conclusions

This paper has provided algorithms to emulate registers and solve consensus under different synchrony assumptions in unknown and anonymous message-passing networks that suffer from crash-failures. One of these algorithms uses a novel pseudo leader election primitive.

Furthermore, we have shown that the MS environment (i.e. a system with a moving timely source) is equivalent to weak-sets, a generalization of registers for unknown and anonymous systems. In some sense, this indicates that the synchrony assumptions in this environment are necessary to implement basic safety primitives.

Additionally, we have shown that in the MS environment, it is not possible to emulate Σ, the weakest failure detector to emulate registers [5], even if we assume the existence of IDs and a bound on the number of processes. To the best of our knowledge, we found for the first time a partially synchronous environment in which registers are implementable and Σ is not.

References

[1] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing omega with weak reliability and synchrony assumptions. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 306–314, New York, NY, USA, 2003. ACM.

[2] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.

[3] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.

[4] Carole Delporte-Gallet and Hugues Fauconnier. Two consensus algorithms with atomic registers and failure detector Ω. In *ICDCN*, LNCS 5408, pages 251 – 262, 2009.

[5] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Shared Memory vs Message Passing. Technical report, LIAFA Paris 7 and EPFL, 2003.

[6] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35:288–323, 1988.

[7] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[8] Fabiola Greve and Sebastien Tixeuil. Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 82–91, Washington, DC, USA, 2007. IEEE Computer Society.

[9] Rachid Guerraoui and Eric Ruppert. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing*, 20(3):165–177, 2007.

[10] M. P. Herlihy and J. M. Wing. Linearizable concurrent objects. In *Proceedings of the 1988 ACM SIGPLAN workshop on Object-based concurrent programming*, pages 133–135, New York, NY, USA, 1988. ACM.

[11] Idit Keidar and Alexander Shraer. How to choose a timing model. *IEEE Trans. Parallel Distrib. Syst.*, 19(10):1367–1380, 2008.

[12] Sara Tucci-Piergiovanni and Roberto Baldoni. Brief announcement: Eventual leader election in the infinite arrival message-passing system model. In *DISC '08: Proceedings of the 22nd international symposium on Distributed Computing*, pages 518–519, Berlin, Heidelberg, 2008. Springer-Verlag.